Refine Search

Search Results -

Terms	Documents
715.clas.	29529

Database: US Pre-Grant Publication Full-Text Database; US Patents Full-Text Database; US OCR Full-Text Database; EPO Abstracts Database; JPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Search:

Refine Search | Recall Text | Clear | Interrupt

Search History

DATE: Wednesday, May 09, 2007 Purge Queries Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L33	715.clas.	29529	L33
L32	715/526	869	L32
L31	715/513	3301	L31
L30	715/500	1470	L30
L29	717/140	977	L29
L28	717/124	1366	L28
L27	717/122	312	L27
L26	717/118	446	L26
L25	717/117	166	L25
L24	717/116	833	L24
L23	717/115	347	L23
L22	717/114	697	L22
L21	717/106	792	L21

L20	717.clas.	13852	L20
L19	707.clas.	43103	L19
L18	707/209	13	L18
L17	707/100	10075	L17
L16	707/2	6395	L16
L15	707/1	9476	L15
DB=EPAB; PLUR=YES; OP=OR			
L14	CN-1361891-A.did.	0	L14
L13	CN-1361891-A.did.	0	L13
L12	CN-1361891-A.did.	0	L12
L11	CN-1361891-A.did.	0	L11
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L10	23919.pn.	2	L10
L9	200023919.pn.	2	L9
L8	20023919.pn.	0	L8
L7	ep-1121654$.did.	1	L7
L6	l1 and (keyword with registry or keyword near registry or keyword adj registry or keyword same registry)	1	L6
L5	l4 and (keyword with registry or keyword near registry or keyword adj registry or keyword same registry)	0	L5
L4	L3 and (java or c or c++ or fortran or cobol)	528	L4
L3	L2 and (program with code or program near code or program adj code)	554	L3
L2	L1 and (extend or extended or expanded or expand)	1021	L2
L1	(macro near language or macro with language or macro adj language)	1912	L1

END OF SEARCH HISTORY

First Hit Previous Doc Next Doc Go to Doc#

End of Result Set

Generate Collection | Print

L6: Entry 1 of 1 File: DWPI Apr 27, 2000

DERWENT-ACC-NO: 2000-364943
DERWENT-WEEK: 200679

TITLE: Extensible macro language providing method for use in computer language processors, involves retrieving code associated with keywords representing new macro command, which is then executed

INVENTOR: DEFFLER, T A; MINTZ, E

PATENT-ASSIGNEE: COMPUTER ASSOC THINK INC (COMPN)

PRIORITY-DATA: 1998US-104682P (October 16, 1998)

Search Selected | Search ALL | Clear

PATENT-FAMILY:

PUB-NO	PUB-DATE	LANGUAGE	PAGES	MAIN-IPC
WO 200023919 A1	April 27, 2000	E	031	G06F017/30
AU 200013152 A	May 8, 2000		000	
EP 1121654 A1	August 8, 2001	E	000	G06F017/30
BR 9914551 A	March 5, 2002		000	G06F017/30
JP 2002528794 W	September 3, 2002		020	G06F009/45
CN 1361891 A	July 31, 2002		000	G06F017/30
AU 772191 B2	April 8, 2004		000	G06F017/30
IL 142564 A	August 1, 2006		000	G06F009/44

DESIGNATED-STATES: AL AM AT AU AZ BA BB BG BR BY CA CH CN CU CZ DE DK EE ES FI GB GE GH GM HU ID IL IS JP KE KG KR KZ LK LR LS LT LU LV MD MG MK MN MW MX NO NZ PL PT RO RU SD SE SG SI SK SL TJ TM TR TT UA UG US UZ VN YU ZW AT BE CH CY DE DK EA ES FI FR GB GH GM GR IE IT KE LS LU MC MW NL OA PT SD SE SL SZ TZ UG ZW AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE

APPLICATION-DATA:

PUB-NO	APPL-DATE	APPL-NO	DESCRIPTOR
WO 200023919A1	October 15, 1999	1999WO-US24115	
AU 200013152A	October 15, 1999	2000AU-0013152	
AU 200013152A		WO 200023919	Based on
EP 1121654A1	October 15, 1999	1999EP-0956567	
EP 1121654A1	October 15, 1999	1999WO-US24115	

EP 1121654A1		WO 200023919	Based on
BR 9914551A	October 15, 1999	1999BR-0014551	
BR 9914551A	October 15, 1999	1999WO-US24115	
BR 9914551A		WO 200023919	Based on
JP2002528794W	October 15, 1999	1999WO-US24115	
JP2002528794W	October 15, 1999	2000JP-0577592	
JP2002528794W		WO 200023919	Based on
CN 1361891A	October 15, 1999	1999CN-0812038	
AU 772191B2	October 15, 1999	2000AU-0013152	
AU 772191B2		AU 200013152	Previous Publ.
AU 772191B2		WO 200023919	Based on
IL 142564A	October 15, 1999	1999IL-0142564	
IL 142564A		WO 200023919	Based on

INT-CL (IPC): G06F 9/44; G06F 9/45; G06F 17/30

RELATED-ACC-NO: 2000-350445;2000-364925 ;2000-364933 ;2000-364941 ;2002-506611 ;2005-182127 ;2006-779548

ABSTRACTED-PUB-NO: WO 200023919A
BASIC-ABSTRACT:

NOVELTY - The providing method involves determining one or more keywords representing new macro command not previously defined in the macro language, in the analyzed macro language expression, based on preset syntax of the macro language. Then, code associated with the keyword is retrieved from registry of keywords and the code associated with keyword is executed.

DETAILED DESCRIPTION - An INDEPENDENT CLAIM is also included for the system for providing extensible macro language.

USE - For providing extensible macro language in computer language processors, word processors.

ADVANTAGE - Extensible macro language is enabled to process the new macro commands, by recognizing the new macro commands unknown to the language and associating the new macro commands with procedure calls stored in registry, thereby allowing dynamic extension of macro language.

DESCRIPTION OF DRAWING(S) - The figure shows block diagram of extensible macro language.

ABSTRACTED-PUB-NO: WO 200023919A
EQUIVALENT-ABSTRACTS:

CHOSEN-DRAWING: Dwg.1/2

DERWENT-CLASS: T01
EPI-CODES: T01-F01; T01-J05B3;

Previous Doc Next Doc Go to Doc#

Refine Search

Search Results -

Terms	Documents
L8 and programming	42

Database:
US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search:

Refine Search

Recall Text Clear Interrupt

Search History

DATE: Wednesday, May 09, 2007 Purge Queries Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L10	l8 and programming	42	L10
L9	l7 and l8	2	L9
L8	(keyword with registry or keyword adj registry or keyword near registry)	84	L8
L7	c near programming	6832	L7
L6	L1 and (keyword with registry or keyword adj registry or keyword near registry)	19	L6
L5	L1 and keyword with registry	19	L5
L4	L2 and keyword with registery	0	L4
L3	L2 and keyword with registry	0	L3
L2	L1 and macro with language with expression	53	L2
L1	c++ or c near programming	38409	L1

END OF SEARCH HISTORY

Macro Processing in High-Level Languages

Alexander Sakharov

Motorola, Inc.
Software Research and Development
3701 Algonquin Rd., Suite 600
Rolling Meadows, IL 60008
sakharov@mot.com

Abstract

A macro language is proposed. It enables macro processing in high-level programming languages. Macro definitions in this language refer to the grammars of the respective programming languages. These macros introduce new constructs in programming languages. It is described how to automatically generate macro processors from macro definitions and programming language grammars written in the lex-yacc format. Examples of extending high-level languages by means of macros are given.

1. Introduction

Traditionally macro processing applies to assembly languages. Applications in assembly languages frequently involve the coding of a repeated pattern of instructions that may contain variable entries at each iteration of the pattern. Macros provide a shorthand notation for these patterns with possible variable fields thereby raising the level of coding. Macro processing requires a separate pass to expand macro calls.

Macros make sense to high-level languages, too. Note that so-called preprocessors possess macro capabilities. For instance, the #define statements in the C preprocessor are basically macros for C [2]. Besides the definition of compile-time parameters, they introduce a shorthand notation. Lisp also allows the definition of macros [20]. M4 in Unix is a general-purpose macro processor [21]. It can be applied to high-level languages.

Macro definitions intended for high-level languages should be specific to their respective base languages. M4 is independent of programming languages. The C preprocessor is independent of C [17]. This independence results in terrible macros like in the following example:

```
#define A(y)   2]y(b
...
d=a [ i * A(*) - c );
...
```

Macros which are specific to the respective programming languages can be used as means for extending the languages. Normally, macros defining extensions which are plain transcriptions are as simple as one-two-three. Even difficult extensions which cannot be expressed as term-rewriting rules for program transformation systems can be expressed with relatively simple macros.

There are several reasons for developing extensions of programming languages. First, there always exists the temptation to build an extended language which is better than a standard language. Any language misses some constructs which look like a promise. Additional constructs may provide a functionality or expressiveness not available in standard languages. Second, some users hate notation of the languages they have to deal with. This happens even when languages are elaborated by ANSI or ISO committees. Third, there exist applications which require a shorter notation than the notation which is available in a standard language. Fourth, availability of extensions may eliminate the need to move on to another language for a new project. It is easier to move on to an extension of a familiar language than to a completely new language.

Currently, adding new constructs into a high-level language is a painful process. It requires changes in compilers. The input languages become incompatible. If divergencies from a standard language are described by macros, and

macro calls are expanded into valid code in the standard language, then a full compatibility of software across all extensions of the standard language is guaranteed. Under such approach, software written in peculiar extensions becomes portable, and free compilers for standard languages can be utilized. Sometimes several languages are simultaneously used in one project. It may be possible to exploit different extensions of one development language in such project instead of putting different programming environments together.

A general-purpose macro language whose macros are relevant to the respective programming languages is proposed in this paper. Connection with the base programming language is done through using references to the grammar of this language in macro definitions. The proposed macro language is very simple. Lex and yacc [12,14,15] specifications describing the syntax of the respective programming language should be given in order to allow macro processing. A macro processor is automatically generated from macro definitions and the lex-yacc specifications.

The generation of preprocessors expanding macros is based on using lex and yacc [15] (or flex and bison [7,19]) and on using a library of C functions supporting symbolic processing. These functions are employed to handle program code represented in the form of trees. Yacc lacks means to handle trees, so does C, the language built in yacc.

2. Macros

Macro usage can be divided into two parts: definition and expansion. A macro definition consists of its header and body. The header is a prototype for the calls of this macro. The header introduces a new syntax in the corresponding programming language. The body is a code in the language, containing formal parameters to be substituted for actual parameters from the calls. The formal parameters are constructs of the programming language. The actual parameters are instances of the constructs. The body also may contain actions which are to be done during expansion. Notice that macro definitions are logically separate from programs containing macro calls.

Expansion consist in replacement of macro calls by macro bodies after parameter substitution. Actions incorporated in macros give a possibility to vary code generated as a result of macro expansion. Due to the actions, macro expansion is a more powerful mechanism than pure rewriting rules. Though, macro definitions are as simple as rewriting rules when actions are unnecessary. There is a difference in semantics between macro processing and program transformation through the use of rewriting rules. Rewriting rules are applied repeatedly whereas macro expansion is performed once for every macro call.

Our macro language comprises two kinds of macro definitions. The first kind of macros has the form:

<grammar_symbol> : <header> ==> <body> ;

The header is a sequence of grammar symbols, literals (characters enclosed in single quotes or strings bracketed in double quotes), or C compound statements. The body is a sequence of grammar symbols, double-quoted literals, pseudo-variables (the dollar sign followed by an integer), or C variables put in parentheses. The syntax of macros has some similarity with yacc grammar productions. Several macros with the same grammar symbol at the beginning can be merged. The vertical bar is used to avoid repeating the symbol. The semicolon should be dropped before the vertical bar. Again, this is similar to the yacc notation.

The parts of these macros till the arrow (==>) represent new grammar productions which will be added to the grammar of the source programming language. The body defines how to expand, i.e. to rewrite the header. Every terminal or nonterminal grammar symbol appearing in the header is a symbol from the original grammar of the source language. Every character enclosed in single quotes from the header is a token of the original grammar. All grammar symbols in the body refer to grammar symbols in the header of a macro. If a grammar symbol occurs more than once in the header, any its occurrence in the body refers to its first occurrence in the header. As in yacc, pseudo-variables refer to the corresponding items of the header. They are used to refer to subsequent occurrences of the same grammar symbol. C variables put in parentheses are from the C compound statements in the header. They should be of type char *, they denote the respective strings.

The other type of macros has the form:

<grammar_symbol> [<action>] ==> <body> ;

Here, optional <action> is a C compound statement, and each item in the body is either the grammar symbol from the left-hand side of this macro, or a double-quoted literal, or a C variable put in parentheses. The bodies of these macros represent the text to substitute for every instance of the grammar symbol in program code. These macros enable very strong extensions. They are not required for defining simple extensions. Normally, macros of this type

are used together with macro definitions of the first type. It is assumed that the bodies in macros of both types depict valid code in the base programming language.

Macro definitions may be preceded by C declarations surrounded in curly brackets. The scope of these declarations is all macros. C variables in parentheses from macro bodies should be defined in this declaration section. A function called stringf may appear in C compound statements from the headers of macro definitions. Its argument is a grammar symbol or pseudo-variable. It yields a string representing the corresponding construct.

Consider a few examples of macro definitions.

A. Default in case statements in Ada

The following simple macro of the first type specifies a new keyword for Ada [4].

```
case_statement_alternative : "default" ';' ==> "when others => null;" ;
```

It gives a shorthand notation for the frequently used text

```
when others => null;
```

B. Iteration over array elements in C++

The next macro incorporates in C++ [8] a new statement for iterating over all elements of an array (the first element through the last element).

```
iteration_statement : "index" '(' IDENTIFIER ';' postfix_expression ')' statement
==> "for(int " IDENTIFIER "=0;" IDENTIFIER "<sizeof(" postfix_expression
")/sizeof(" postfix_expression "[0]);" IDENTIFIER "++)" statement ;
```

According to this macro, the code

```
index(i;a) a[i]=0;
```

is translated into the following valid C++ code:

```
for(int i=0;i<sizeof(a)/sizeof(a[0]);i++) a[i]=0;
```

C. Finite state machines in Pascal

A construct which may be convenient for programming finite state machines is introduced in Pascal [11] by means of several macros. The macros below turn a finite state machine notation into conditional statements occurring within a repetitive statement. It is assumed that variables statevar and eventvar are predefined in programs which contain finite state machine specifications. More efficient implementations of finite state machines like one from [13] can be given by more complex macros.

```
RepetitiveStatement :
    "start" ConstantIdentifier "event" Expression "stop" ConstantIdentifier BEGIN TransitionList END
==> "begin statevar:=" $2 ";"
"repeat eventvar:=" Expression ";\n" TransitionList "\nuntil statevar=" $6 "end" ;

TransitionList :
    Transition ==> Transition |
    TransitionList Transition ==> TransitionList ";\n" Transition ;

Transition :
    '(' ConstantIdentifier ',' ConstantIdentifier ')' "-->" ConstantIdentifier CompoundStatement
==> "if statevar=" $2 "and eventvar=" $4 "then \n\tbegin statevar:=" $7 ";" CompoundStatement "end" ;
```

BEGIN and END denote the corresponding tokens. Consider an example of a finite state machine [13] encoded in the proposed notation:

```
start off event button() stop off begin
(off,onoff)-->on begin toggle() end;
(off,volp)-->on begin toggle(); volume(1) end;
(on,onoff)-->off begin toggle() end;
(on,volp)-->on begin volume(1) end;
(on,volm)-->on begin volume(-1) end end
```

This code is transformed into the following code in Pascal:

```
begin statevar:=off; repeat eventvar:=button();
if statevar=off and eventvar=onoff then
    begin statevar:=on; begin toggle() end end;
if statevar=off and eventvar=volp then
    begin statevar:=on; begin toggle(); volume(1) end end;
if statevar=on and eventvar=onoff then
    begin statevar:=off; begin toggle() end end;
if statevar=on and eventvar=volp then
    begin statevar:=on; begin volume(1) end end;
if statevar=on and eventvar=volm then
    begin statevar:=on; begin volume(-1) end end
until statevar=off end
```

D. Lambda expressions in C

The two macros below serve to allow lambda expressions [3] in C [2]:

```
{ int lam=0; int plam=0; char * lamidty[N]; char * lamid[N]; char lamfty[N];
char * lamexp[N]; char * lamf[N]; char * curlf; char * funcdefs; }
unary_expression : "lambda" '(' type_name ')' IDENTIFIER ':' '(' type_name ')' unary_expression
{ lamidty[lam]=stringf($3); lamid[lam]=stringf(IDENTIFIER); lamfty[lam]=stringf($8);
lamexp[lam]=stringf(unary_expression); sprintf(strptr,"_lf%d_\0",lam);
curlf=lamf[lam]=strptr; strp+=strlen(strptr)+1; lam++; }
==> (curlf) ;
```

The unary expression and identifier in the header of the above macro are bound by the lambda quantifier. Only global variables may occur in unary expressions bound by the lambda quantifier. These lambda expressions contain extraneous cast subexpressions attached to the identifier bound by lambda and to the expression to which lambda applies. It is done in order to simplify macros describing this extension. This restricted class of lambda expressions [3] is broad enough for practical needs. Notice that lambda expressions are present in functional languages but missing from imperative languages. The ordinal number of the last lambda expression is stored in variable lam. Variable plam keeps the highest value of lam before parsing the current function definition. Arrays lamidty, lamid, lamty, lamexp, lamf serve for storing pieces of program code. Variable strptr points to the currently available location in some pool (see the next chapter).

A new function definition is created for every lambda expression. It is done by the following macro of the second kind:

```
function_definition
{ int i; funcdefs=strptr; strcpy(strptr,"");
for (i=plam;i<lam;i++)
    { strcat(strptr,lamfty[i]); strcat(strptr," "); strcat(strptr,lamf[i]);
    strcat(strptr,"("); strcat(strptr,lamidty[i]); strcat(strptr," _lv_) {");
    strcat(strptr,lamid[i]); strcat(strptr,"=_lv_; return "); strcat(strptr,lamexp[i]);
    strcat(strptr,"; }\n"); strptr+=strlen(strptr)+1; }
plam=lam; }
==> (funcdefs) function_definition ;
```

Let us encode the problem of computing the double integral:

$$e = \int_{a}^{b}\int_{c}^{d} f(x, y)\, dy dx$$

Suppose a function called itgl calculates definite integrals. Its parameters are: a continuous function having one floating-point argument and returning floating-point values; two bounds which are floating-point values. This problem is encoded with lambda expressions as follows:

```
e=itgl(lambda (float) x:(float) itgl(lambda (float) y: (float) f(x,y),c,d),a,b);
```

This assignment is transformed into the C assignment:

```
e=itgl(_lf1_,a,b);
```
Two new function definitions are generated from the assignment with lambda. They will precede the function in which the lambda expression occurred.

```
float _lf0_ ( float _lv_) { y = _lv_; return f(x,y); }
float _lf1_ ( float _lv_) { x = _lv_; return itgl(_lf0_,c,d); }
```

3. Building Macro Processors

Yacc and lex specifications of a source programming language should be given in order to enable macro processing. These specifications describe the syntax of the language. They should not include any translation actions. For LALR(1) languages [1], these yacc rules are bare rules without actions. The yacc specifications may include actions that embody context analysis that may be required to parse the programming language. The typedef names in C [2] is an example of such non-context-free construct.

The most natural way to make macro expansion in accordance to the syntax of a high-level language is to build trees representing expanded code. Leaves of these trees represent tokens in the source language or text patterns from the bodies of macro definitions. Every interior node corresponds to a nonterminal from either the source grammar or macro definitions. For valid code in the base language, these trees are parse trees. Given a language grammar written in the lex-yacc style and macros transformed into additional yacc rules, such trees can be built by providing each definition in all yacc rules with the action that yields the corresponding tree. The code of these actions can be automatically constructed and inserted into yacc rules. Another possible way to perform macro expansion is to process strings representing programs. Representing programs as trees is advantageous because it does not require to move long strings in memory.

Two tasks are done to enable macro expansion. First, original lex and yacc specifications defining the syntax of the source programming language are automatically modified. Second, macro definitions are turned into additional grammar productions which are incorporated with the source yacc specifications of the respective programming language. Besides, the names of some grammar symbols are changed, and new rules are added to the original lex specifications of the base language.

Some requirements are established and restrictions are imposed on the original yacc and lex specifications in order to simplify their automatic modification. White spaces should be collected by lex rules in an external variable called whsp. This variable is cleared up to a null string after a token is returned. The line number should be stored by lex rules in an external variable called linecnt. Variable yylval [15] should not be used in the specifications. Pseudo-variables $$, $1, $2, etc. should not appear in the original yacc specifications. The %union and %type declarations should not appear in the yacc specifications. Variable strptr of type char * may be used in C statements from macro definitions. It points to the currently available location in a global pool. Data may be placed there. The value of strptr should be increased to refer to a free space, then.

The actions that construct trees are automatically inserted in the original yacc rules defining the source programming language. Each generated action returns the corresponding tree through the $$ pseudo-variable [15]. Values of pseudo-variables $1, $2, etc. and values stored at yylval are trees, too. The trees are represented in memory as Lisp lists [6,20]. Token representations and white spaces from parsed code are passed to yacc rules through the yylval variable. A call of function gtoken is automatically inserted into each lex rule in order to form a required value of yylval. For example, the lex rule

```
"<=" { return LE; }
```

is transformed into:

```
"<=" { gtoken(); return LE; }
```

Consider an Ada [4] grammar production written for yacc:

```
if_statement : IF condition THEN sequence_of_statements END IF ;
```

Here, IF, THEN, and END denote the corresponding tokens. An action is generated and then added to this rule; it becomes:

```
if_statement : IF condition THEN sequence_of_statements END IF
    { $$=glist($1,$2,$3,$4,$5,$6,N!L); } ;
```

Here, glist is a function from our library for symbolic computations in C. This function constructs and yields the list

of all its parameters as items except the last parameter which should always be NIL [6]. The action

```
{ $$=gstring(""); }
```

is added to null right-hand sides of grammar productions. The function gstring generates an atom [6] from its parameter of type char *.

Besides, an additional grammar rule is generated for any base programming language. It introduces a new start symbol which is defined through the original one. The role of this rule is to emit expanded code. For instance, the following rule is generated for the Pascal grammar [11]:

```
Program_start : Program { emit($1); } ;
```

Program_start becomes the start symbol instead of Program. The function emit traverses the tree that is its parameter and emits strings assigned to leaves.

Macro definitions of the first kind are transformed into additional grammar rules which are incorporated with the modified yacc specifications of the programming language. Macro definitions containing double-quoted literals in their headers also cause the generation of new token definitions which are to be added to the modified lex specifications of the programming language. The headers of the macros of the first kind constitute these additional grammar rules. Double-quoted literals in the headers are replaced by the names of the new tokens generated from the literals. The actions associated with these additional grammar productions are intended to convert instances of constructs defined by macro headers into instances of the corresponding macro bodies. These automatically generated actions transform macro calls into trees representing program code in the source language.

In actions generated from macros, function glist is applied to all constituents of the macro body. Grammar symbols from macro bodies are changed for the corresponding pseudo-variables ($1,$2,...). The pseudo-variables point to the occurrences of the grammar symbols in the respective headers which become rule definitions. Function gstring is applied to all literals and C variables from bodies.

The aforementioned macro definition of a new kind of iteration statements in C++ is transformed into the yacc rule:

```
iteration_statement : TOKEN1 '(' IDENTIFIER ',' postfix_expression ')' statement
    { $$=glist(gstring("for(int "),$3,gstring('=0;"),$3,gstring("<sizeof("),$5,
    gstring(")/sizeof("),$5,gstring("[0]);"),$3,gstring("++)"),$7,NIL); } ;
```

Besides, the following rule is added to the modified lex specifications of C++:

```
"index" { gtoken(); return TOKEN1; }
```

For each macro definition

```
<grammar_symbol> [ <action> ] ==> <body> ;
```

of the second kind, all occurrences of <grammar_symbol> in the left-hand sides of grammar rules defining the source programming language are transformed into <grammar_symbol>_aux. The grammar rule

```
<grammar_symbol> : <grammar_symbol>_aux <additional_action> ;
```

is added to the modified yacc specifications. The additional action is generated from the body. These actions are similar to those built for the macros of the first kind. If a macro of the second kind contains an action, then the generated action begins with the code of that action.

For instance, the aforementioned macro definition

```
function_definition { ... } ==> (funcdefs) function_definition ;
```

is turned into the yacc rule:

```
function_definition : function_definition_aux { ... $$=glist(gstring(funcdefs),$1,NIL); } ;
```

All occurrences of function_definition in the left-hand sides of grammar productions defining C are replaced by function_definition_aux.

When all specifications generated from macro definitions are added to the modified lex and yacc specifications of the source programming language, lex and yacc are run with these newly created specifications which incorporate the syntax of the source language and extensions given by macros. Then, C code generated by lex and yacc is compiled, and a working program is obtained after linkage with other object files [15]. The working program performs macro expansion by building trees whose leaf nodes refer to pieces of expanded code, traversing the trees, and emitting the program code assigned to leaf nodes. Flex [19] and bison [7] can be utilized to handle extended grammar specifications instead of lex and yacc for achieving a higher speed of macro processing.

Two programs serve to modify specifications of the source programming language. One of them modifies the lex specifications of the language. The other transforms the yacc specifications. They are separated from the program that creates preprocessors because modifications of the lex and yacc specifications of the source language need to be done once for any given language. We use lex and yacc to parse and modify the yacc specifications of the source programming language. The syntax of yacc specifications in the form of yacc rules is given in [12]. The program that modifies the lex specifications of the source language is implemented in C.

The syntax of macro definitions is written in the lex-yacc format. Macro definitions are parsed and transformed into the form of additional lex and yacc specifications by means of lex and yacc, too. The technique exploited to turn macro definitions into yacc rules is the same as that for transforming code in language extensions into code in standard languages. Trees which represent the additional specifications are generated with using functions glist and gstring in yacc actions from the syntax description of our macro language. Then, these trees are traversed, and the textual representation of the additional specifications is emitted.

Yacc generates LALR(1) parsers [1]. Since yacc is utilized to generate macro processors, the expansion is a LALR(1) translation. Macros should not spoil LALR(1) properties of the base language. It is a common problem in creating new languages or in extending well-defined languages. Literals in double quotes from macro definitions become new tokens. They should not make language ambiguous either.

4. Concluding Remarks

Programming language extensions are easily defined with macro definitions. Compilers do not have to be changed to digest languages extended by means of macros. Instead, preprocessors that turn code in an extended language into code in a standard programming language are built automatically. Compare this approach with building compilers for extended languages [10]. High-level language macro processors may be helpful for constructing preprocessors that translate among standard variants of one programming language.

Macros look like rewriting rules. Though, there is a difference in semantics between macro processing and program transformation through the use of rewriting rules [5]. Rewriting rules are applied repeatedly whereas macro expansion is performed once. Macros provide a shorthand notation. Rewriting rules may describe, for example, optimizations which are applied until no improvement is achieved. As shown by examples, the proposed class of macros is suitable for defining extensions which depend on context or affect it. Generally speaking, such extensions cannot be specified by pure rewriting rules. Capabilities of C are employed through yacc actions in order to help define such extensions. Though, extensions which are simple transcripts are defined in a declarative fashion.

Author's first attempts to embody the idea of macro processing in high-level languages were based on using MetaTool [16]. They failed because of problems with fitting in the MetaTool's limitations. Switching to lex, yacc, and a library of C functions for symbolic processing merely simplified implementation. Our implementation approach is similar to that of J. Park [18]. He extended the syntax of yacc rules. Additional expressions attached to yacc rules are removed in the process of translation, and new actions are generated and inserted in the rules.

There are some open issues about macro processing in high-level languages. One of the aims of this report is to solicit suggestions from others on macros for high-level languages. Our language of macro definitions is influenced by the language of yacc specifications. It would differ if it were intended to work with other parser generators [9]. Is it worth designing a macro language which is independent of parser generators?

Yacc specifications of a standard programming language need some error reporting and recovery code. Employing a parser generator with automatic error recovery [9] instead of yacc could simplify the task of writing syntax specifications of programming languages. Compilers that are invoked after preprocessing report errors in terms of expanded code, so do symbolic debuggers. It is feasible to set references to original code through the #line command for C or C++ as a base programming language. There is no general solution of error reporting for preprocessed code. Line numbering may be preserved in expanded code, but constructs differ.

C and C++ are equipped with preprocessors by definition [2,8]. Most C implementations run the preprocessor separately from the compiler. Though, the ANSI C standard [2] does not require that the preprocessor operates as a separate pass. If the two are tied, then it is not clear how to incorporate a macro processor.

References

[1] Aho, A.V., Sethi, R., Ullman, J.D., *Compilers. Principles, Techniques, and Tools*, Addison-Wesley Publ. Co., 1986.

[2] *American National Standard for Information Systems, Programming Language C*, ANSI X3.159-1989, American National Standard Institute, Inc., NY 1990.

[3] Barendregt, H.P., *The Lambda Calculus: Its Syntax and Semantics*, Elsevier, 1981.

[4] Barnes, J.G.P., *Programming in Ada plus Language Reference Manual*, Addison-Wesley Publ. Co., 1991.

[5] Boyle, J.M., Abstract Programming and Program Transformations - An Approach to Reusing Programs, Software Reusability, v. 1, Addison-Wesley Publ. Co., 1989, 361-413.

[6] Charniak, E., Riesbeck, C.K., McDermott, D.V., Meehan, J.R., *Artificial Intelligence Programming*, Lawrence Erlbaum Associates, 1987.

[7] Donnelly, C., Stallman, R., *Bison Reference Manual*, Free Software Foundation, 1988.

[8] Ellis, M., Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley Publ. Co., 1990.

[9] Grosch, J., Generators for High-Speed Front Ends, *Lecture Notes in Computer Science*, v. 371, 1989, 81-92.

[10] *High C Language Extensions Manual*, MetaWare Inc., 1990.

[11] Jensen, K., Wirth, N., *Pascal User Manual and Report*, Springer-Verlag, 1991.

[12] Johnson, S.C., *Yacc - Yet Another Compiler-Compiler*, AT&T Bell Laboratories, 1975.

[13] Kuuluvainen, I., Vanttinen, M., Koskinen, P., The Action-State Diagram: A Compact Finite State Machine Representation for User Interfaces and Small Embedded Reactive Systems, *IEEE Transactions on Consumer Electronics*, v. 37, 1991, #3, 651-658.

[14] Lesk, M.E., Schmidt, E., *Lex - A Lexical Analyzer Generator*, AT&T Bell Laboratories, 1975.

[15] Mason, T., Brown, D., *lex & yacc*, O'Reilly & Associates, 1990.

[16] *MetaTool Specification-Driven-Tool Builder*, User Manual, AT&T, 1990.

[17] Mischel, J., Dusting off COBOL, *Computer Language*, v.8, 1991, #11, 41-46.

[18] Park, J.C.H., y+: A Yacc Preprocessor for Certain Semantic Actions, *SIGPLAN Notices*, v.23 (1988), #6, 97-106.

[19] Paxson, V., *Flex - Manual Pages*, 1990.

[20] Steele, G., *Common Lisp: The Language*, Digital Press, 1984.

[21] *UNIX Programmer's Manual*, AT&T Bell Laboratories, 1983.

A Model of Extensible Language Systems

Presentation given to the International Symposium on Extensible Languages.

Grenoble
September 1971

M G Notley
Scientific Centre
IBM (UK) Ltd
Neville Road
Peterlee
Co Durham

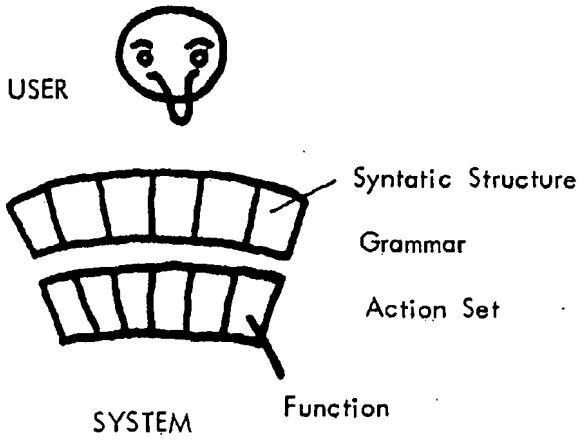
1. Introduction

At the present time the subject of extensible languages appears to suffer from the lack of any central coherent framework to knit together the many pieces of individual work that are being done. This paper is an attempt, therefore, to fill that lack.

What appears to be required is some central conceptual model or paradigm of languages and language extensibility onto which framework we can hang all these pieces of work.

This paper, therefore, contains of four main points:-

* The description of a very simple model of formal machine languages and their operation.
* A discussion of the generality of this model.
* A discussion of the usefulness of this model for the study of extensible languages.
* An illustration of the application of the model by reference to an on-going implementation at the IBM (UK) Scientific Centre.



2. Basic Model

Consider first a simple imperative language system in which the user has a set of simple commands, each one of which causes a single action.

In this case there exists a set of basic functions which the machine may perform, and which we may call the action set. Mapping onto these functions (in a one-to-one or many-to-one mapping) is a set of basic syntactic structures (in this case simple commands) which we may call the grammar of the language available to the user.

3. Local Syntax

Usually, however, things are not as simple as this. The concept of a local syntax must be introduced since, in general, the basic functions of the action set are really families of functions, and further syntax is required to discriminate between different members of a given family. Let me give you an example. Suppose one of the available functions is to display a histogram of a previously identified set of data on a visual display screen. Then further syntactic structures must exist to determine, for example, the scales of the axes and the bin width of the histogram.

Clearly such syntactic structures are required in the language, and clearly they are only required locally (in the context of a histogram in this example). One can imagine, then, the command "HISTOGRAM" causing some local section of syntax to become activated. This local syntax will then retain control until either an error is found or else the required parameters of the associated function are determined. In this latter case the function is then invoked, in the interpreted system, or else the appropriate compiled code is stored. After the local syntax has done its job, control then returns to the previous global syntax. In this model, then, all context sensitivity is assumed to be handled purely by the invocation of local syntaxes.

This local syntax need not, of course, be a simple one-to-one mapping. For example, default values and synonyms may exist, and conversational techniques might be used to prompt the user about neglected parameters. Also, the idea of a local syntax is not to be confused with block structure. The former is a property of the programming language used and the functions it addresses, whereas block structure is a property of a particular programme in that language and controls the environment of a particular operation.

Also of interest is the way in which context sensitivity is handled by this model. Clearly the global language is context sensitive, but each block of local syntax is context free internally and could, for example, be stated in BNF. This raises the possibility of handling context sensitivity by an extension of the BNF notation that reflects the concept of a local syntax. There is, however, an assumed implication hidden here that would have to be investigated. This assumption concerns the essentially functional incidence of any requirement for context sensitivity.

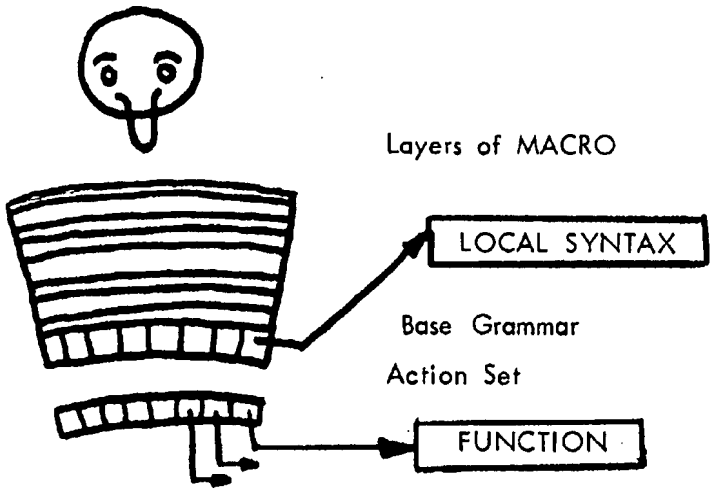
4. Macro

The imperative command language described so far may not be very user friendly, or convenient. To handle this problem it is, in general, necessary to introduce, at the global level, the use of macros.

A macro consists simply of an instruction to replace certain parts of the input text by new text. It acts at the global level and is context free. The important point to recognise now is that all macro replacements are completed at the global level before control is passed to the local syntax. Indeed, in a purely context free language it would be possible to perform all the required activity in terms of the macros alone.

5. Basic Building Block

The basic building block of the model is now, therefore, as follows:



The users input text passes through the top skin of macros to a basic grammar. This grammar maps onto an action set, and with each function of this action set is associated (at least potentially) a local syntax. This block therefore acts, in a sense, just like a multi-way switch. A complex input text is ultimately transofrmed into the invocation of one or a sequence of the functions from the action set.

One could think of this switching as occurring at many different levels. For example one might want to think of the input text causing, ultimately, a switching between difference basic machine instructions. On the other hand the identical system could be thought of as switching between different systems routines which are the basic actions of some hypothetical or actual interpreter.

For example, one might write a query language whose action set is a set of N compiled PL/I routines. These PL/I routines, in turn, might be considered as an input text to the PL/I switching mechanism whose action set is the IBM System/360 instruction set.

The concern is not with interpretation or compilation, but if these PL/I routines were actually interpreted, as in the PL/I checkout compiler, one could think of the system as follows. The original query, in the query language passes through the global macros, through the local syntax and a PL/I function is invoked. This PL/I function, as an actual string of PL/I text, then itself passes through global macros, local syntax, and eventually a number of 360 machine instructions are invoked.

6. The Idea of a "Cut"

To handle this complexity, when thinking about a given system, it is of advantage to pin down exactly where one is thinking of this switching occurring. This concept will be referred to as a "cut". A cut may be thought of as a set of actions, possible at different levels in the system, such that all possible activity is included and none duplicated. In the query language, the most convenient cut would probably be at the level of the set of PL/I macros. However, in considering one particular function in detail one might wish to drop to a lower level by replacing that function by a number of lower level alternative sub-functions, whilst leaving all other actions the same.

This idea of a cut does not however necessarily imply anything about either the binding time or the compilation/interpretation balance of the system. For any particular cut one could always construct a real system below which all is iterpreted and above which all is compiled, but one is really talking about the theoretical system which would be simulated by any such real system.

7. Generality of the Model

The generality of this model is best considered in terms of some of the more significant properties of formal languages:-

* Action Set
    * Storage of commands
        * Conditionals
        * Procedures
    * Storage of Data
        * Data types

Let us look first at data types. The uses of data types are simply these: to allow arrays and structures to be handled as single objects, and to allow the use of polymorphic operators. Thus if I "add" an array of floating-point numbers to an integer a whole lot of computation goes on behind the scenes with which I no longer need to concern myself. This allows me, as a user, to consider my language as acting at a higher level cut. Clearly, then, the mechanism of data types must be handled, in my model, in the local syntax sections of those functions of the action set that must act differently for different data types.

Two alternative techniques could be considered. Every object name could have appended to it some textual indication of its data type at the global macro level, for later recognition by the local syntax. Alternatively, and acting at a different cut, a declaration in the global syntax could be thought of as changing (in the sense of further restricting) the global syntax so that the named object could legally be used in future in the context "integer add" but not "floating-point add", for example.

Now let us look at procedures. It is a truism to say that to write a new procedure is to extend, in a certain sense, the base language. Let us take a look at what is actually happening here. If I write a new procedure and give it a name, ALPHA, say and arguments A, B and C, then I now have a new function available to me in the total system. This function is the function ALPHA with a local syntax <A>, <B>, <C> . Now it is quite clear, in this example, why it depends on my point of view whether or not I consider this to be a language extension. If I am the programmer, then I am thinking of the system at a lower cut, and thus no extension has occurred but simply a new, complex, macro which changes the invocation of ALPHA to a number of lower level invocations. If, however, I am not the programmer but some relatively unsophisticated user then I see the system at a much higher level cut. In this case a new function has been added to my system and I see it as a genuine extension.

It appears , then, that a model of this type will be sufficiently general to encompass most of the interesting features of formal machine languages.

8. Usefulness of the Model

There are three main uses of this model. The first of these is simply the normal advantages that accrue from conceptual simplicity.

A good programming language could be defined as a language that allows the prgrammer to do the things he wants to do, and in which he does not make mistakes'. The only way of ensuring the programmers do not make mistakes, apart from insisting on a great deal of manual reading, is to provide them with a language with the least number of surprises. Things happen, in general, as they expect them to happen. This is only possible if the programmer and the language designer share a common simple concept of the language.

In the field of extensible languages this is going to be, in some ways, more difficult. However, I believe that this model, is sufficiently simple and general to point to ways of solving this problem. With this model it is clear, for example, that there are three possible extension mechanisms:-

* New macro
* New or changed local syntax
* New function

- the last of which cannot as I will demonstrate, be a system facility as such. For each of these functions, quite simple paradigms can be developed to provide a common conceptual framework for the programmer and systems designer.

The second way in which this model is useful is the way in which it exposes some interesting design decisions for extensible language systems.

For example, we have discussed the different cuts at which one could view the same system. Now in most practical systems, there is usually some level below which it is not required to provide a built-in extension function. One could always solder on new hardware, for instance, but it would hardly be practicable or desirable to cause this to be automatic. Thus, for a given extensible language system there is a design decision required as to the lowest cut at which language extensions are to be allowed. Once this lowest cut has been decided, effort may then be put into making the lower end of the system really efficient at the sacrifice of unnecessary flexibility. Any changes below that level may now be referred to as genuine function extensions and, though the system may be designed to be able to receive them easily, such changes cannot be specified in the top level language itself.

This choice of the lowest cut of an extensible language system must obviously be made by weighing the flexibility of a low-level lowest cut against the cost in efficiency and implementation. Careful consideration is also required as to the interpreter/compiler balance of the implemented system.

Another design decision exposed by this model is the decision as to where, amongst three major alternatives, the complexity and implementation effort is to be distributed. The system, whether extensible or not, could be implemented with the major complexity embodied in:-

* A large number of local, locally-context-free syntaxes, no macros and a simple keyword oriented command language (as in the system we are currently implementing).

* Many successive layers of macro replacement, no local syntaxes, and a large low-level action set (as in the Markov algorithm approach).

* No macros or local syntaxes but very complex invocation structures from one syntax block to another so that the compiler itself becomes a dynamic parsing tree with backtrack on error.

When extensible language systems are considered it becomes of critical importance which approach or combination of approaches is used depending on the type and efficiency of the language extension mechanisms proposed.

The third way in which this model is useful is the way in which it can guide our research into extensible languages. One result, for example, that appears to follow from this model is that only two language extension mechanisms are required (apart from actual function extensions) in any extensible language system.

These mechanisms must be able to handle respectively:-

New macros
New or changed local syntaxes

It appears that these two mechanisms are both necessary and sufficient. To be able actually to prove this, with a reasonable degree of rigour, would be a significant step forward in the general study of extensible language systems.

9. An On-going Implementation

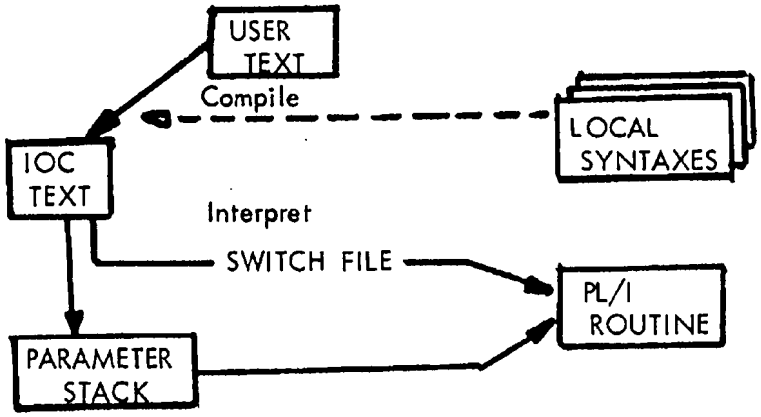
In order to demonstrate that these ideas are actually useful for the design of an extensible system, an on-going implementation we are engaged in at the Scientific Centre will be described briefly.

The system is a general purpose information system which is intended to be "customisable". That is, there is a core system with a very simple basic query language. This core system, however, may be extended by function extensions and syntax extensions to be tailored to any particular application.

For reasons of practicality, the lowest cut, for language extension purposes, consists of an action set which is a set of compiled PL/I subroutines. These subroutines are invoked by the user by simple keyword commands.

There is a compilation phase during which the user input text is transformed, with the help, if necessary, of interactive prompting, into a compact intermediate object code which may be stored or executed immediately.

This intermediate object code consists simply of coded subroutine parameters to be stacked (such as "bin width" for a histogram routine) and subroutine calls, coded as index numbers to a file of sub-routine entry points termed the switch file.



The interpreter, which is the central control point of the system, is thus very simple. It takes intermediate object code text and whenever it encounters a parameter it stacks it, and whenever it encounters a switch file number, it passes control to the indicated entry point.

The system, however, is designed to facilitate function and syntax extensions, and can therefore become quite sophisticated.

A function extension consists of writing a PL/I procedure, testing it and debugging it. A local syntax for that procedure is then written by the same programmer using a very simple "PL/I - like" macro language. The new function is then inserted into the system by putting into the switch file a triple:-

* The function name
* The local syntax entry point
* The procedure entry point

This extension can be carried out by any moderately competent PL/I programmer with no special knowledge of the total system.

A syntax extension consists, at present, only of the ability to alter or extend local syntaxes and no macro facility exists. This will be the subject of some later work. Until this enhancement is completed, the user language must thus apparently always remain a keyword oriented command language with the present system.

However, even with this restriction a lot of sophistication could be achieved if required, since one of the facilities available in the local syntax is the embedding of the top level language. Thus clearly, an effective global syntax with macros could be achieved by embedding all possible syntax blocks in one local syntax, (that of the command "LOGON" for example!)

It should be noted that this system makes clear and absolute, the distinction between syntax , as embedded in the entry point names and local syntaxes, and function as embedded in the compiled PL/I procedures. This has many advantages for an extensible system. For example, it would be quite possible to change the whole user language from English to French without having to scrap or re-design any of the compiled procedures!

As a final comment on this system it may be of interest that early next year an experimental version will go into use. Swedish ecological scientists are currently collecting data to create a research information system of up to some five million bytes for the study of the Ecology of the Baltic seaboard.